Notations for Algorithms

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

September 12, 2017

Abstract

A quick guide to the notations used in the course.

This is a guide to the programming notations used by me in the course. It is meant to be explanatory rather than prescriptive; you are certainly welcome to use other notations as long as they are clear or clearly explained.

1 Notations for algorithms

1.1 Assignment commands

Assignment commands have the form $V := E$, where V is a variable and E is an expression.[1] For example

$$x := x + 1$$

1.2 Sequential composition

As illustrated above, sequential composition is indicated by putting one command after another.

$$x := y \; y := t$$

[1] Some programming langauges use $V = E$. I prefer to use $=$ for equality. Regardless of how you write it, please pronounce it "V becomes E" or "V is assigned E."

Usually we stack sequentially composed commands vertically,

$$x := y$$
$$y := t$$

but there is no reason to do so.[2] We can add semicolons between or after commands if we like

$$x := y; y := t \qquad \text{or} \qquad x := y; y := t;$$

1.3 Variable and constant introduction

Local variables are introduced by a command of the form **var** $V : T := E \cdot C$ where V is a variable name, T is a type, E is an expression and C is a command; C is the scope of the variable. For example

$$\textbf{var } t : \text{Int} := x \cdot x := y;\ y := t$$

The prefix **var** $V : T := E \cdot$ has lower precedence than sequential composition, so the command

$$\textbf{var } t : \text{Int} := x \cdot x := y;\ y := t$$

groups as

$$\textbf{var } t : \text{Int} := x \cdot (x := y;\ y := t)$$

The type and colon can be omitted when the type can be inferred or isn't important.

$$\textbf{var } t := x \cdot x := y;\ y := t$$

The expression and the := can be omitted when no initialization is needed.[3]

$$\textbf{var } t \cdot\ t := x; x := y;\ y := t$$

When a variable is never changed I use **val** instead of **var**, to emphasize that the variable represents a single value. (Java uses the keyword **final** for a similar purpose. Dafny uses **let**.) For example

$$\textbf{val } t := x \cdot x := y;\ y := t$$

[2] Other than to improve readability; which is a pretty good reason.

[3] It's very rare that a variable needs to be introduced before it can be sensibly initialized.

Sometimes the initial value of variable doesn't need to be nailed down to a specific value. For example if we just need that q be initialize to a number whose square is bigger than s, we could write:

$$\textbf{val}\ q \mid q^2 > s \cdot ...$$

The vertical bar can be read as "such that". For another example suppose, we need to break a set S into two subsets that are disjoint and not empty; we introduce two new variables like this

$$\textbf{val}\ T, U \mid S = T \cup U\ \wedge\ T \cap U = \emptyset\ \wedge\ T \neq \emptyset \neq U \cdot ...$$

1.4 Alternation

The choice between two alternatives is given by a command of the form

$$\textbf{if}\ E\ \textbf{then}\ C_0\ \textbf{else}\ C_1\ \textbf{end if}$$

The expression E should be Boolean. For example

$$\textbf{var}\ x \cdot \textbf{if}\ y > z\ \textbf{then}\ x := y\ \textbf{else}\ x := z\ \textbf{end if}\ \cdots$$

When nothing needs to be done if E is false, we write

$$\textbf{if}\ E\ \textbf{then}\ C\ \textbf{end if}$$

Sometimes there are more than 2 alternatives:

$$\textbf{if}\ E\ \textbf{then}\ C_0\ \textbf{elseif}\ E_1\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{end if}$$

abbreviates

$$\textbf{if}\ E\ \textbf{then}\ C_0\ \textbf{else if}\ E_1\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{end if end if}$$

and so on.

Sometimes it is useful to use alternation in expressions.

$$\textbf{if}\ E_0\ \textbf{then}\ E_1\ \textbf{else}\ E_2\ \textbf{end if}$$

For example the declaration and initialization of x above could be written as

$$\textbf{var}\ x := \textbf{if}\ y > z\ \textbf{then}\ y\ \textbf{else}\ z\ \textbf{end if} \cdot\ \cdots$$

1.5 Switch commands

Here is an example switch command

switch E
case F **then**
    // This code is executed if $E = F$
    some code
case G, H, I **then**
    // This code is executed if $E = G$, $E = H$ or $E = I$.
    some other code
else
    // This code is execute otherwise
    the is the default code
end switch

Some differences with the C-style (or Java-style) switch

- There is no fall-through, so no break command is needed to prevent it.
- Expressions need not be constants, i.e., they may depend on variables. This is illustrated in the next example.
- If the "**else**" is missing, it's an error for no case to apply.
- If more than one choice applies, it is a nondeterministic (i.e., arbitrary) choice as to which is chosen.

As an example of these differences, consider

switch *true*
case $a \leq b$ **then** $b := b - a$
case $a \geq b$ **then** $a := a - b$
end switch

In cases where $a = b$, both case expressions will equal true, and so either can be picked. The choice could be resolved at run time or before run time. For example when realizing the pseudocode of the last example to C or Java it would be reasonable to realize it as

```
if( a <= b ) {
```

```
   b = b - a ; }
else {
   a = a - b ; }
```

It would be possible —though probably not the most sensible thing to do— to implement it as

```
if( a < b || a==b && a%2 == 0 ){
   b = b - a ; }
else {
   a = a - b ; }
```

Nondeterministic choice is a form of abstraction. It allows us to document that the choice between two alternatives inessential to the algorithm's design and so it allows the reader to concentrate on what is important. Here is another example

while $|S| > 1$ **do**
  val $T, U \mid T \cup U = S \wedge T \neq \emptyset \wedge U \neq \emptyset$
  switch *true*
  case $T \cap G \neq \emptyset$ **then** $S := T$
  case $U \cap G \neq \emptyset$ **then** $S := U$
  end switch
end while

Here when $T \cap G \neq \emptyset$ and $U \cap G \neq \emptyset$ are both true, it is inessential which assignment to S is picked.

1.6 Iteration (loops)

A while loop expresses an iteration that happen an indefinite number of times. While commands have this form

while E **do** C **end while**

Often we know how many times a loop needs to be repeated. I'll write

for $V \leftarrow [j, ..k]$ **do** C **end for**

to mean that C is repeated $k - j$ times[4] with variable V taking on the values

[4]The exception is when $k - j$ is negative, then the loop is repeated 0 times.

j, $j+1$, ..., $k-1$. (Note the absence of k!) More generally we can write

for $V \leftarrow E$ **do** C **end for**

where E is an expression of type sequence or set. When E is a sequence, C is executed once for each item of the sequence, starting with the first item and ending with the last. When E is a set, C is executed once for each element of the set in any order.

1.7 Assertions

Assertions are Boolean expressions that indicate what we expect will be true at various points in the execution of an algorithm. I'll write assertions in curly braces. For example

$$\{x = X \wedge y = Y\}$$
$$x := x + y$$
$$\{x = X + Y \wedge y = Y\}$$
$$y := x - y$$
$$\{x = X + Y \wedge y = X\}$$
$$x := x - y$$
$$\{x = Y \wedge y = X\}$$

As a special case, an assertion that precedes a loop is that loop's invariant and should be true at the start and end of each iteration. For example the following annotation is not valid

var $i := 0$·
var $s := 0$·
$\{i = 0 \wedge s = 0\} \longleftarrow$ not correct
while $i < n$ **do**
    $s := a(i)$
    $i := i + 1$
end while

A valid annotation would be

var $i := 0$·

var $s := 0\cdot$
$\left\{0 \leq i < n \wedge s = \sum_{k \in \{0,..i\}} a(k)\right\}$
while $i < n$ **do**
    $s := a(i)$
    $i := i + 1$
end while

1.8 Sequences and sets

For each type T, seq $\langle T \rangle$ is the type of one-dimensional sequences with items from T. For example

$$\textbf{var } a : \text{seq } \langle \text{Int} \rangle := [2, 3, 5, 7, 11]\cdot \tag{1}$$

declares a to be a variable that holds a sequence of values of type Int and initialized the variable to a particular sequence of length 5. Sequence values (like all values) are immutable. For example

var $a := [2, 3, 5, 7, 11]\cdot$
var $b := a\cdot$
$a := [2, 3, 9, 7, 11]$

Would assign b the value $[2, 3, 5, 7, 11]$ and a the value $[2, 3, 9, 7, 11]$.

My discrete math review notes provide notations for sequences. For example $a(2)$ is the third item of a and $a[0, ..n]$ is the sequence of the first n items.

We can write $a(2) := 9$ to mean replace the third item. I.e., $a := a[0, ..2]\hat{}[9]\hat{}a[3, ..a.\text{Length}]$.

1.9 Procedures and functions

Procedures are written as

proc $p(x : T, \textbf{var } y : U)$
    C
end p

Here

- p is a procedure name
- x is a value parameter of type T
- y is a **var** parameter of type U that may be changed (thus changing the corresponding argument).
- C is a command

Arguments for var parameters must be variables. The value of the argument changes when the procedure returns.

Functions are written as

func $f(x : T) : U$
  C
end f

Here

- f is the name of the function.
- The function returns a value of type U
- C is a command that may contain "return" commands

1.10 Contracts

Preconditions and postconditions define the contract of a procedure or function. I usually write them like this

proc $p(x, y : \mathbb{N}, \mathbf{var}\ z : \mathbb{N})$
pre $x > 0 \land y > 0$
post $z|x \land z|y \land \forall w \in \mathbb{N} \cdot w|x \land w|y \Rightarrow w|z'$
  C
end p

Since x and y are not marked var, they won't change and so the mentions of x and y in the postcondition refer to their initial value (which is the same as the final value). For var parameters and global variables, the value might

change, and we use the notation old(x) to refer to the value of the variable (or any expression) when the procedure started. For example

proc enbiggen(**var** $x : \mathbb{N}$)
post $x >$ old(x)
    C
end enbiggen

The postcondition asserts that the value of x at the end of any invocation of enbiggen will be greater than the value x had at the start of that invocation.

For functions, I use the special name *result* in postconditions to represent the result of the function. E.g.

func between($x, y : \mathbb{N}$)
pre $y - x > 1$
post $x < result < y$
    return $\left\lfloor \frac{x+y}{2} \right\rfloor$
end between

1.11 Generic procedures

Generic procedures and functions follow Java's conventions. E.g., a function that operates on any two sets whose elements are the same type and returns the same type of set might start with.

func symdiff $\langle E \rangle$ $(S, T : \text{set} \langle E \rangle) : \text{set} \langle E \rangle$

1.12 Classes

Here is an example

class C
    private var *f0* : *T0*
    private var *f1* : *T1*
    public proc *m0*()
        S0

end *m0*
public func *m1*() : *T2*
S1
end *m1*
end *C*

Generic classes

class $C\,\langle T\rangle$
...
end C

Here C is a class name and T is a type variable.
For example $C\,\langle \text{Int}\rangle$ would be a class in which T is replaced by Int in C.

1.13 Indentation and optional ends

Like real code, pseudocode should be properly indented.

Sometimes to save space on the board or in the notes, I leave off the **end**s from constructs and let the indentation define the structure, like this

proc bsearch$\langle T\rangle$(**var** $S : \text{set}\,\langle T\rangle\,, G : \text{set}\,\langle T\rangle$)
pre $S \cap G \neq \emptyset \wedge |S| \in \mathbb{N}$
post $S \subseteq G \wedge |S| = 1$
$\{S \cap G \neq \emptyset \wedge |S| \in \mathbb{N}\}$
while $|S| > 1$ **do**
val $T, U \mid T, U \subset S \wedge T \cup U = S \wedge U \neq \emptyset \neq T \cdot$
switch true
case $T \cap G \neq \emptyset : S := T$
case $U \cap G \neq \emptyset : S := U$

However, when you do this in hand writing or across page-breaks, it's good to use vertical lines to show how the commands line up

proc bsearch$\langle T\rangle$($S : \text{set}\,\langle T\rangle\,, G : \text{set}\,\langle T\rangle$)
pre $S \cap G \neq \emptyset \wedge |S| \in \mathbb{N}$
post $S \subseteq G \wedge |S| = 1$
| $\{S \cap G \neq \emptyset \wedge |S| \in \mathbb{N}\}$

```
    while |S| > 1 do
        val T, U | T, U ⊂ S ∧ T ∪ U = S ∧ U ≠ ∅ ≠ T·
        switch true
        case T ∩ G ≠ ∅ : S := T
        case U ∩ G ≠ ∅ : S := U
```